PATENT
5681-80700
P9307

"EXPRESS MAIL" MAILING LABEL NUMBER EL990142769US
DATE OF DEPOSIT JANUARY 29, 2004
I HEREBY CERTIFY THAT THIS PAPER OR FEE IS BEING DEPOSITED WITH THE UNITED STATES POSTAL SERVICE "EXPRESS MAIL POST OFFICE TO ADDRESSEE" SERVICE UNDER 37 C.F.R. §1.10 ON THE DATE INDICATED ABOVE AND IS ADDRESSED TO THE COMMISSIONER FOR PATENTS, BOX PATENT APPLICATION, P.O. Box 1450, ALEXANDRIA, VA 22313-1450

Derrick Brown

DYNAMIC DISTRIBUTION OF TEST EXECUTION

By:

Yaniv Vakrat and Victor Rosenman

B. Noël Kivlin
Meyertons, Hood, Kivlin, Kowert & Goetzel
P.O. Box 398
Austin, TX 78767-0398

Dynamic Distribution of Test Execution

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims the benefit of Provisional Application No. 60/443,795. This application is related to Application No. (STC File No. 47979), filed on even date, entitled *Parallel Test Execution on Low-End Emulators and Devices*.

BACKGROUND OF THE INVENTION

1. Field of the Invention.

[0002] The present invention relates generally to hardware and software testing and verification, and specifically to testing software on low-end emulators and computing devices.

2. Description of the Related Art.

[0003] The meanings of acronyms and certain terminology used herein are given in Table 1:

Table 1

API	Application programming interface
CLDC	Connected, limited device configuration. CLDC is suitable for devices with 16/32-bit RISC/CISC microprocessors/controllers, having as little as 160 KB of total memory available.
HTTP	HyperText Transfer Protocol
ID	Identifier
IP	Internet Protocol
J2EE	Java 2 Enterprise Edition
J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edition
JAD	Java application descriptor
JAR	Java archive
JDTS	Java Device Test Suite Execution Framework
MIDlet	A MIDP application
MIDP	Mobile information device profile. A set of Java APIs, which, together with the CLDC, provides a complete J2ME application runtime environment targeted at mobile information devices.

[0004] MIDP is defined in Mobile Information Device Profile (JSR-37), JCP Specification, Java 2 Platform, Micro Edition, 1.0a (Sun Microsystems Inc., Palo Alto, California, December 2000). MIDP builds on the Connected Limited Device Configuration (CLDC) of the Java 2 Platform, Micro Edition (J2ME) (available from Sun Microsystems Inc., Palo Alto, California). The terms Sun, Sun Microsystems, Java, J2EE, J2ME, J2SE, and the Sun logo are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States of America and other countries. All other company and product names may be trademarks of their respective companies. A portion of the disclosure of this patent document contains material that is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by any one of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.

[0005] Tools have been developed in recent years to aid in the design verification of hardware and software systems, for example software suites, hardware circuitry, and programmable logic designs. In order to assure that the design complies with its specifications, it is common to generate a large number of input or instruction sequences to assure that the design operates as intended under a wide variety of circumstances. In general, test systems produce a report indicating whether tests have been passed or failed, and, in some cases may even indicate a module that is estimated to be faulty.

[0006] Conventionally, in order to test a device under development (such as a mobile information device), or to test software designed to run on such a device, a developer connects the device to an appropriate test system. The target device under test may be connected to the test system either directly or

via a communication emulator. The developer selects a battery of test programs to run on the target device while monitoring its behavior. Running the complete battery of tests can commonly take many hours or even days. This problem is particularly acute in testing low-end computing devices, such as cellular telephones and other mobile information devices, which have limited computing power and memory resources. Thus, testing on the target device can become a serious bottleneck in the development cycle.

[0007] A centralized system for centrally managing test suites is disclosed in commonly assigned Application No. (STC File No. 47900), entitled "Automated Test Execution Framework with Central Management", which is herein incorporated by reference. In this arrangement, a central repository contains a management unit, available test suites and a single test execution harness or framework. Using the management unit, a system administrator establishes active versions of the various test suites and their individual configurations. End users install clients of the central repository, using a system-provided installer program. In each client, an execution script is created, which downloads the harness and a local configuration file. Then, when the harness is executed at the client, it loads with all designated test suites already installed, configured and ready for execution. The client always has the most current versions of all test suites. Advantageously, all necessary information is obtained from a single central location.

[0008] A further improvement in test suite management is disclosed in commonly assigned Application No. (STC File No. 47979), entitled "Parallel Test Execution on Low-End Emulators and Devices", which is herein incorporated by reference. While this arrangement facilitates testing large numbers of devices simultaneously, it does not optimally reduce the time re-

quired for the verification phase of the development cycle. This is particularly the case where a developer wishes to exhaustively test a device or software program using a very large number of different tests.

SUMMARY OF THE INVENTION

[0009] Embodiments of the present invention provide methods and systems for parallel testing of multiple low-end computing devices, such as mobile information devices in which different tests in a test suite are dynamically allocated to different instances of a computing device-under-test. Multiple instances of a computing device-under-test are connected to a test server, either directly or via an emulator. Each of the devices is assigned a unique identifier (ID), which allows the server to keep track of which tests have been assigned to and carried out by each device. Whenever a device completes a test (or a bundle of tests), it reports the results to the server and requests the next text to execute, using its ID in the messages that it sends to the server. Based on the ID and the report, the server selects the next test or test bundle to assign to this device. This mechanism enables the server to dynamically distribute different elements of a test suite to the different instances of the computing devices-under-test, and to balance and track the load of testing among an arbitrarily large number of client devices. The distribution of the load is optimized for completion of the test suite in minimal time. According to the invention, the realistic possibilities of failure of particular instances of the computing device-under-test, and the availability of additional computing devices in the course of the verification process are accommodated. The load of tests is dynamically redistributed to a changing population of computing devices-under-test, that is while tests are cur-

rently running in the devices. Upon completion of the tests, the execution framework aggregates the results returned by all of the devices-under-test into a single set of results, and produces a consolidated report. With this arrangement, the test suite can be completed in far less time than is required by test systems known in the art. For example, dividing a test suite of 1000 tests among four devices can reduce the run time by up to a factor of four.

[0010] The invention provides a method for testing computing devices, which is carried out by providing a suite of test programs on a server for execution by a plurality of the computing devices that are coupled to the server, distributing different ones of the test programs from the server to the computing devices for concurrent execution thereof, receiving messages from the computing devices upon completion of the different test programs, and responsively to the messages, iteratively distributing the test programs until all of the test programs in the suite have been executed.

[0011] According to an aspect of the method, the test programs are distributed as JAR files and JAD files.

[0012] According to an additional aspect of the method, the JAD files are constructed responsively to the messages.

[0013] Another aspect of the method includes coupling a new computing device to the server, and reallocating the test programs to the computing devices and the new computing device.

[0014] One aspect of the method includes detaching one of the computing devices from the server, and marking unexecuted ones of the test programs that were distributed to the detached computing device as "not run".

[0015] In a further aspect of the method the different test programs are distributed by removal of the test programs from a stack.

[0016] Yet another aspect of the method distributing includes assigning the different test programs in groups, so as to minimize completion time of the suite.

[0017] The invention provides a computer software product, including a computer-readable medium in which computer program instructions are stored, which instructions, when read by a computer, cause the computer to perform a method for testing computing devices, which is carried out by accessing a suite of test programs on a server for execution by a plurality of the computing devices that are coupled to the server, distributing different ones of the test programs from the server to the computing devices for concurrent execution thereof by the computing devices, receiving messages from the computing devices upon completion of respective the different ones of the test programs, and responsively to the messages, iteratively distributing remaining test programs until all of the test programs in the suite have been executed.

[0018] The invention provides a method for testing computing devices, which is carried out by providing a suite of test programs on a server for execution by a plurality of the computing devices that are coupled to the server, assigning a respective unique identifier to each of the plurality of the computing devices for use in communicating with the server, making respective allocations of different ones of the test programs for the computing devices, downloading the allocations from the server for respective execution by the computing devices coupled thereto, so that at least first and second computing devices among the plurality execute different first and second test programs from the suite substantially simultaneously. The method is further carried out by receiving messages at the server from the computing devices with respect to the execution of the test programs, each of the messages containing

the respective unique identifier, and responsively to each of the messages, downloading at least another of the test programs to a respective one of the computing devices.

[0019] According to an aspect of the method, the computing devices comprise MIDP-compliant devices, and wherein the test programs comprise MIDlets, which are packaged in respective JAD files and JAR files, and wherein allocating the test programs includes downloading the JAD files and the JAR files to the MIDP-compliant devices.

[0020] The invention provides a computer software product, including a computer-readable medium in which computer program instructions are stored, which instructions, when read by a computer, cause the computer to perform a method for testing computing devices, which is carried out by accessing a suite of test programs that are stored on a server for execution by a plurality of the computing devices that are coupled to the server, assigning a respective unique identifier to each of the computing devices, for use in communicating with the server, making respective allocations including different ones of the test programs for the computing devices, downloading the allocations from the server for respective execution by the computing devices coupled thereto, so that at least first and second computing devices among the plurality execute different first and second test programs from the suite substantially simultaneously, receiving messages at the server from the computing devices with respect to the execution of the test programs, each of the messages containing the respective unique identifier, and responsively to each of the messages, returning a new allocation of unexecuted ones of the test programs to respective ones of the computing devices for execution thereof.

[0021] The invention provides a server for testing computing devices, including a communication interface for cou-

pling a plurality of the computing devices thereto, and a processor having instructions to access a suite of test programs for execution by the computing devices that are coupled to the server, and to distribute at least a portion of different ones of the test programs via the communication interface to respective ones of the computing devices for concurrent execution thereof. The processor has further instructions to receive messages via the communication interface from the computing devices indicating completion of the test programs, and responsively to the messages, to distribute remaining ones of the test programs to the computing devices for execution thereof iteratively until all of the test programs in the suite have been executed.

[0022] The invention provides a server for testing computing devices, including a communication interface for coupling a plurality of the computing devices thereto, and a processor having instructions to access a suite of test programs for execution by the computing devices that are coupled to the server, to assign a respective unique identifier to each of the plurality of the computing devices for use in communicating with the server, to make respective allocations including different ones of the test programs for the computing devices, to download the allocations from the server for respective execution by the computing devices coupled thereto, so that at least first and second computing devices among the plurality execute different first and second test programs from the suite substantially simultaneously. The processor has further instructions to receive messages from the computing devices indicating completion of the execution of the test programs, each of the messages containing the respective unique identifier, and responsively to the messages to distribute remaining ones of the

test programs iteratively to the computing devices for execution thereof.

BRIEF DESCRIPTION OF THE DRAWINGS

[0023] For a better understanding of the present invention, reference is made to the detailed description of the invention, by way of example, which is to be read in conjunction with the following drawings, wherein like elements are given like reference numerals, and wherein:

[0024] Fig. 1 is a block diagram that schematically illustrate systems for parallel testing of low-end computing devices, in accordance with an embodiment of the present invention;

[0025] Fig. 2 is a block diagram that schematically illustrate systems for parallel testing of low-end computing devices, in accordance with an alternate embodiment of the present invention;

[0026] Fig. 3 is a block diagram that schematically illustrates program components used in a test system, in accordance with an embodiment of the present invention;

[0027] Fig. 4 is a flow chart that schematically illustrates a method for parallel testing of low-end computing devices, in accordance with an embodiment of the present invention; and

[0028] Fig. 5 is a flow chart illustrating a method of dynamic test load distribution in accordance with a disclosed embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

[0029] In the following description, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent to one skilled in the art, however, that the present invention may be

practiced without these specific details. In other instances well-known circuits, control logic, and the details of computer program instructions for conventional algorithms and processes have not been shown in detail in order not to unnecessarily obscure the present invention.

[0030] Software programming code, which embodies aspects of the present invention, is typically maintained in permanent storage, such as a computer readable medium. In a client/server environment, such software programming code may be stored on a client or a server. The software programming code may be embodied on any of a variety of known media for use with a data processing system, This includes, but is not limited to, magnetic and optical storage devices such as disk drives, magnetic tape, compact discs (CD's), digital video discs (DVD's), and computer instruction signals embodied in a transmission medium with or without a carrier wave upon which the signals are modulated. For example, the transmission medium may include a communications network, such as the Internet.

[0031] Reference is now made to Fig. 1, which is a block diagram that schematically illustrates a system 20 for parallel testing of multiple mobile information devices 24, in accordance with an embodiment of the present invention. The system 20 is built around a test server 22, which is described in greater detail hereinbelow. The devices 24 are client devices, and are typically low-end devices, with limited computing power and memory, for example, cellular telephones or personal digital assistants (PDA's). In the description that follows, the devices 24 are assumed to comply with MIDP, but the principles of the present invention are equally applicable to other types of low-end computing devices, operating in accordance with other standards and specifications. The server 22 typically comprises a programmable processor, and has suitable

communication interfaces, such as wireless or wired interfaces, for communicating with multiple devices 24 simultaneously.

[0032] Each of the devices 24 receives a unique identifier for communicating with the server 22. Typically, the unique identifier may comprise a unique Internet Protocol (IP) address that is assigned to each of the devices 24 for communicating with the server 22. Alternatively, the server may assign IDs of other types, or the ID's may be assigned by a user upon initiating communication between one or more of the devices 24 and the server 22. Methods for assigning and using these IDs are described in detail hereinbelow.

[0033] Reference is now made to Fig. 2, which is a block diagram that schematically illustrates a system 30 for parallel testing of multiple devices 24, in accordance with another embodiment of the present invention. In this embodiment, the server 22 communicates with the devices 24 through a test host 32, such as a personal computer or workstation. Multiple test hosts of this sort may be connected to the server 22 in parallel, but only a single host is shown in Fig. 2 for the sake of simplicity. The host 32 can simultaneously accommodate multiple devices 24, but the host 32 typically has only a single IP address. Therefore, in this embodiment, the IP address cannot be used conveniently to identify the individual devices 24, and an alternative unique identifier is typically used, as described below.

[0034] Reference is now made to Fig. 3, which is a block diagram that schematically illustrates software program components running on the server 22 and the devices 24, in accordance with an embodiment of the present invention. Elements of this software may be provided to the server 22 and to the devices 24 on tangible media, such as optical or magnetic storage media or semiconductor memory chips. The software may be

downloaded to the server 22, and alternatively or additionally, to the devices 24 in electronic form, for example, over a network or over the air.

[0035] The server 22 comprises a test framework 40, which generates and deploys the tests to be carried out by the devices 24. The test framework 40 may be implemented as the "Java Device Test Suite" execution framework (JDTS) (version 1.0 or higher), available from Sun Microsystems, Inc., which employs MIDP. A suitable version of the test framework 40 is described, for example, in the above-mentioned Application No. (STC File No. 47900), which is commonly assigned herewith, and is herein incorporated by reference.

[0036] The tests typically are packaged in the form of Java applications contained in a set of JAD and JAR files. Each JAR file of this sort, together with its accompanying JAD file, is referred to hereinbelow as a test bundle 52. Users of the system 20 (Fig. 1) or the system 30 (Fig. 2) interact with the test framework 40 in order to select the tests to be executed by the system. Alternatively, other test frameworks may be used for generating the required test files, as will be apparent to those skilled in the art.

[0037] A test manager 42 in the server 22 is responsible for serving requests from the devices 24, based on the unique client identifiers mentioned above. Typically, whenever one of the devices 24 makes a request, the test manager 42, typically operating as a main thread, reads the request and assigns a new thread 44 to handle it. This thread 44 retrieves the client unique identifier from the request, calls the components of the test framework 40 that are needed to process the request, and then returns the appropriate response to the client device, as described hereinbelow. After assigning the thread 44 to handle the client, the main thread of the test

manager 42 waits for the next client request. Each client request is handled by a separate thread 44, which terminates upon completion of processing. This arrangement, together with the unique identifier mechanism, ensures that the server 22 will be able to handle multiple devices 24 simultaneously without confusion.

[0038] In order to run Java applications, the devices 24 contain an implementation of the Connected Limited Device Configuration specification, CLDC 46, with an implementation of the Mobile Information Device Profile specification, MIDP 48, running over the CLDC 46. The applications that run on this technology, such as the tests supplied by framework 40, are known as MIDlets. These applications are created by extending an API MIDlet class of the MIDP 48. Thus, each test bundle 52 is actually a MIDlet, packaged in the form of a JAD/JAR file pair.

[0039] The test bundle 52 is typically downloaded to the devices 24 in a two-step process:

[0040] 1. The server 22 downloads the JAD file, which contains environment settings and some environment demands. Application Manager Software, AMS 50, which is typically a part of a browser built into the devices 24, evaluates the JAD file to ensure that the device is able to accept the MIDlet. For example, the JAD file for a given MIDlet may specify that the device must support MIDP version 2.0. If the device does not support this version, the AMS 50 rejects the application download, and saves the time that would otherwise be consumed by downloading the much larger JAR file.

[0041] 2. After completing all the relevant checks, the AMS 50 reads from the JAD file the loca-

tion of the corresponding JAR file on the server 22 and asks to download the JAR file to one or more of the devices 24. The JAR file contains all the relevant classes of the test bundle 52.

[0042] Once the JAR file for the test bundle 52 is downloaded to one of the devices 24 and stored in the local device memory, the device is ready to run the tests of the test bundle 52. Every JAR file that the AMS 50 downloads to the devices 24 typically contains an agent 54, which is used to run the tests, in addition to classes corresponding to the tests themselves. To start test execution the AMS 50 runs the agent class. The agent 54 then addresses the server 22 in order to receive instructions regarding the next test to run (getNextTest) and to report test results (sendTestResult), typically using a protocol based on HTTP. Each test in the test bundle 52 corresponds to a respective class in the JAR file. Each client request that is addressed by the agent 54 to the server 22 includes the unique identifier that has been assigned to the particular one of the devices 24, so that the server 22 is able to recognize the client and serve it in the correct manner.

Implementation Details.

[0043] Further details of the implementation of the server 22 are given in Listing 1 (class BaseHttpServer). An implementation of the communications interface through which requests and messages are transmitted between the server 22 and the devices 24 is detailed in Listing 2 (class Communicator). Runtime generation of JAD files by the server 22 is accomplished using Listing 3 (class HttpServer). Launching of the agent 54 is detailed in Listing 4 (class MIDPRunner). Implemen-

tation of the thread 44 is detailed in Listing 5 (class ServerTaskThread).

[0044] Listing 6 shows a class (class Extender) that is needed by the classes shown in Listings 1 - 5. A brief description of Listing 6 follows.

[0045] A public interface Extender provides access to a class Extender. The class Extender enables an agent link with platforms that require extensions of their main application class, for example to properly employ a system class, such as class Applet or class MIDlet. The class Extender accepts delegation of platform specific commands from an agent.

[0046] The interface Extender includes the following methods. A method getRunnerExtender retrieves a reference to a platform class, which the main application class extends. Using this method, an agent provides access to the test program by the main application class in the context in which it is currently executing. An object is returned, which can be cast to the system class that the extender class extends. A method terminateAgent provides a platform-specific way of application termination.

[0047] It will be understood that Listings 1 - 6 are exemplary, and that the functions and operations shown therein can be accomplished using other techniques known to the art.

[0048] Reference is now made to Fig. 4, which is a high level flow chart that schematically illustrates a method for running test suites on multiple client devices 24 in the system 20 (Fig. 1) or the system 30 (Fig. 2), in accordance with an embodiment of the present invention. For clarity of presentation, the flow chart in Fig. 4 presents an interaction involving only a single client request. However, the method can be performed simultaneously, with many clients. Indeed, different devices may be executing different tests, or even different

test suites or test bundles at any given time. This method is explained with reference to the software structures shown in Fig. 3, although other implementations are also possible, as will be apparent to those skilled in the art. The method begins at initial step 100, which is a configuration step. A server is attached to a plurality of client devices to be tested using suitable communications links.

[0049] Next, at delay step 102 the server awaits a request from a client. As will be apparent from the discussion below, the request could be for a new test bundle, or for the next test in a test bundle that is currently executing.

[0050] Upon receipt of a client request, control proceeds to decision step 104. Here it is determined whether the client request received at delay step 102 is a request for a new test bundle. This is normally the case when the client is first recognized by the server. Otherwise, such a request can occur if a previous test bundle has been completed by a client already known to the server according to its unique identifier.

[0051] If the determination at decision step 104 is negative, then generally, the server is already aware of the requesting client. Control proceeds to decision step 106, which is disclosed below.

[0052] If the determination at decision step 104 is affirmative, it is concluded that the server has not previously interacted with the requesting client. Control proceeds to step 108. Here a unique identifier is assigned to the requesting client. Whichever of the alternate methods disclosed herein for making the assignment is employed, the client is uniquely identified at step 108, and its subsequent requests and results will be handled without possibility of confusion with other currently attached clients. As noted above different clients may be identically configured, and may even be concurrently

executing the same test bundle. Furthermore, any test results reported by the now uniquely identified client are accurately associated with that particular client so as to guarantee the integrity of test reports that may be eventually generated by the server. Control now proceeds to step 110.

[0053] At step 110 a JAD file corresponding to the client request is generated or selected by the server for transmission to the client. Control then proceeds to step 112, which is disclosed below.

[0054] Decision step 106 is performed when the determination at decision step 104 is negative. Here it is determined if the client request received at delay step 102 is a request for the next test to be executed in a current test bundle. Such requests may be generated at the client responsively to evaluation at the client of a previously downloaded JAD file, based on suitability of the tests for the particular client.

[0055] If the determination at decision step 106 is affirmative, then control proceeds to step 114. The server retrieves the test record that corresponds to the next test to be executed by the client. It will be apparent that this mechanism provides a high degree of central control by the server, so as to optimize the order of test execution by different clients. For example, if the server has received borderline test results from the client, it could elect to repeat a particular test, or to perform supplemental tests that would otherwise be skipped.

[0056] If the determination at decision step 106 is negative, then it is concluded that an unrelated client request has been made. For example, the client may have requested transmission of test results, or a display illustrating the profile of a test. Control proceeds to step 116, where this request is processed.

[0057] Next, at step 112 a response to the client request is assembled. Test information obtained in step 114, a JAD file obtained in step 110, or information relating to the request processed in step 116, whichever is applicable, is now concatenated with the client's unique identifier.

[0058] Next, at step 118, the response assembled at step 112 is downloaded to the requesting client. Control then returns to delay step 102, where the next client request is awaited.

[0059] Further details of the method shown in Fig. 4 are disclosed in copending commonly assigned Application No. (STC File No. 47979), entitled *Parallel Test Execution on Low-End Emulators and Devices*, which is herein incorporated by reference.

Dynamic Load Distribution.

[0060] As disclosed above, and with continued reference to Fig. 3, the test framework 40 maintains a dynamic record of the correspondence between test bundles and clients (devices). Therefore, whenever a particular client sends a request, e.g., a request to provide another test, the test framework 40 responds by accessing the JAR file containing the appropriate next test of a current test bundle for that client. Further details of the server-side operation are shown a code fragment in Listing 7.

[0061] The number of tests in a test bundle is user configurable, and the size of each JAR file may be limited by the user. Thus, the number of tests downloadable to the client in a single JAR file is also limited, both by the size restriction imposed by the user on the JAR file, and by the constant overhead of the agent 54, which is typically about 20K, and which must be included in the JAR file. For instance, if the

user has limited the JAR file to 60K, then 20K are preempted by the agent 54, leaving 40K available for test classes, and for whatever additional classes may be required to support execution of the test classes. During configuration of a session of the test framework 40, as many JAR files are created as may be needed to contain all the tests of a given test suite, taking all the above-noted factors into consideration, and maintained on a stack. When a request is received from a client, the test framework 40 associates the next element on the stack with the requesting client, and creates an appropriate JAD file on-the-fly, which is then downloaded to the requesting client as explained hereinabove. Assignment of the tests to be performed in the JAD file takes into consideration the number of clients connected to the test framework 40 in order to equitably distribute the test load among all the clients upon request. Alternatively, various algorithms may be used to accomplish this. For example, the tests may be allocated by number. Alternatively, the tests may be weighted according to the time required to perform each of them, and the tests then distributed according to weighted scores. For example, two tests each requiring one hour to perform could be assigned to a first client using a first JAD file. Four tests each requiring 1/2 hour to perform could be assigned to a second client using a second JAD file. Both clients would complete their assignments in the same two-hour interval, but the second client would perform twice as many tests as the first client. Further implementation details are provided in the code fragment of Listing 8.

[0062] It will also be recalled that whenever a new client is connected to the server 22, a unique ID is assigned to the client. Additionally, the test framework 40 recognizes the new client as being connected. The user may dynamically add as many devices as he wishes during the course of the session,

while tests are running on currently attached devices. The test framework 40 creates a stack of test bundles, and associates a test bundle with each new client.

[0063] It is also possible to dynamically remove a client during the course of the session, even while the client is engaged in test execution. This can be accomplished simply by physically disconnecting it from the server 22. However, because a JAR file is already associated with this client by a unique ID, no other device can complete the incomplete tests included in that particular JAR file (other than an illegal client, that is a client that has been improperly recognized under the identifier of the previously disconnected client). At the test framework 40, the test executing at the time the client was removed is marked by a special code (VM_EXIT). Upon completion of the session all unexecuted tests in the JAR file will have been marked as "not run", unless the client is reconnected in order to complete the tests in the test bundle.

[0064] Reference is now made to Fig. 5, which is a flow chart illustrating a method of dynamic test load distribution in accordance with a disclosed embodiment of the invention. The method begins at initial step 120, in which a test framework prepares test bundles for execution. JAR files are configured, taking into account the memory considerations disclosed above.

[0065] Control now proceeds to decision step 122, where it is determined if a new client has been connected to the test framework. If the determination at decision step 122 is negative, then control proceeds to decision step 124, which is disclosed below.

[0066] If the determination at initial step 120 is affirmative, then control proceeds to step 126. Here a unique identifier is assigned to the new client as disclosed above.

[0067] Next, at step 128, test assignments are prepared for the new client. A JAD file is prepared, which includes the tests of a test bundle that are to be performed. The test framework here takes into account the number of tests that have been selected for each client to run, and the client's configuration, in order to most equitably distribute the test load.

[0068] Next, at step 130 the JAD file prepared in step 128 and a corresponding JAR file are downloaded to the client, which proceeds to execute the tests as disclosed above. Control then returns to decision step 122.

[0069] Decision step 124 is performed if the determination at decision step 122 is negative. Here it is determined if a client that has been connected to the test framework has been disconnected, or has gone off-line. If the determination at decision step 124 is negative, then control returns to decision step 122.

[0070] If the determination at decision step 124 is affirmative, then control proceeds to step 132. Here the test framework marks the client as having exited. If the client is not subsequently reconnected during the session, then upon completion of the session, the test currently being performed by the disconnected client and all other tests assigned to the disconnected client will be noted as not having been run. Control now returns to decision step 122.

[0071] It will be appreciated by persons skilled in the art that the present invention is not limited to what has been particularly shown and described hereinabove. Rather, the scope of the present invention includes both combinations and subcombinations of the various features described hereinabove, as well as variations and modifications thereof that are not in the prior art, which would occur to persons skilled in the art upon reading the foregoing description.

COMPUTER PROGRAM LISTINGS

Listing 1

```
public abstract class BaseHttpServer implements Runnable {

  /**
  * "Runnable" is a standard Java class.
  *
  * Current tests is a hash table, which hold current tests
  * for each midp client
  * according to its signature.
  */
  private Hashtable currentTests  =  new Hashtable ();


  /**
  *The class that manage the distribution of the
  * bundles to the clients,
  * and dealing with requests that related to the
  * bundles such as get next test.
  */
    public TestProvider getTestProvider() {
        return testProvider;
    }

  /**
  *Set the current test of a specific client.
  */
    public void setCurrentTest
      (String signature, byte[] args)
    {
      currentTests.put (signature, args);
    }

  /**
  *Get the current test of a specific client.
  */
    public byte[] getCurrentTest(String signature)
  * {
      byte[] currentTest =
  *      (byte[]) currentTests.get (signature);
      return currentTest;
    }

  /**
  *Remove the current test of a specific client.
  */
```

```
    public void removeCurrentTest(String signature){
        currentTests.remove (signature);
    }


    public void setTestProvider(TestProvider tp) {
        testProvider = tp;
        mainClass = tp.getAppMainClass();
       // The MIDlet name

        if (mainClass == null) {
            throw new NullPointerException
             ("Provider main class not defined");
        }
        jarSourceDir = tp.getJarSourceDirectory();
           //the directory of the JAR files.
        if (mainClass == null) {
            throw new NullPointerException
             ("Provider jar source directory not defined");
        }
    }



  public void run() {
    Socket conn = null;
    aborted = false;
    started = true;
    synchronized(this) {
     notify();
    }

    while (!done) {
      try {
        try {
          conn = socket.accept();
          //retrieve the IP from the connection
          String ipAddress  =
             conn.getInetAddress().getHostAddress();

          //assign new thread to process
          // the client request.
          ServerTaskThread task =
            new ServerTaskThread(ipAddress, conn, this);

          //process request data
          task.processRequest ();

          //start that task.
```

```
            task.start ();
          }
          catch (InterruptedIOException iie) {
               continue;
               }
          }
          catch (Exception x) {
            x.printStackTrace();
          }
        }

        synchronized (socket) {
           try{
             aborted = true;
             LOCALHOST = false;
             nextID = 1;
             socket.close ();
             socket.notifyAll();
           }
           catch (IOException ioe){
             ioe.printStackTrace ();
           }
        }
        }



        /** reads alternative JAM tags if "jam.tags"
         * property file provided
         * Keys:
         * AppURL - Application URL JAM Tag
         * AppSize - Application Size JAM Tag
         * MainClass - Application Main Class JAM Tag
         * AppName - Application Name JAM Tag
         * AppUseOnce - Use-Once JAM Tag
         */
        public abstract void loadTagNames();


       /** Runtime generation of JAD file, to be sent
        * to the client as result of
        * getNextApp request.
        */
        public abstract ByteArrayOutputStream
          generateJAM (String host,
                       int port,
                       String next_app,
                       long length,
```

```
                String mainClass,
                String jarLocation,
               String protocolPermissions,
               String agentMonitorID,
               String signature,
               // attaching the client ID to the JAD file.
                 ServerTaskThread task)
                   throws UnsupportedEncodingException;


  /**
  * In case that a certain ID is already in use,
  * than the server assign an ID to the client.
  */
    public int getNextID(){
        return nextID++;
    }

  /**
  * When an assigned thread was completed its work,
  * we need to remove it from the lsit of threads
  * currently running in the system.
  */
    public void cleanUp(String signature){
        synchronized(taskHashtable){
            taskHashtable.remove (signature);
        }
    }

  /**
  * When this class assign new thread for processing
  * the client request, it puts
  * the thread in a hashtable for keeping reference
  * to it. If a former thread,
  * that serve a particular client, has not completed
  * yet, the former thread is
  * terminated (generally this case happens
  * when something went wrong).
  */
    public void putTaskThread (String signature,
     ServerTaskThread serverTaskThread){
        synchronized(taskHashtable){
            ServerTaskThread task =
               (ServerTaskThread)getTaskThread(signature);
            if (task != null){
                verboseln("task is being stoped");
                task.stopTask();
                task = null;
            }
```

```
                taskHashtable.put(signature,
                  serverTaskThread);
            }
          }

     /**
      *Get the reference to a client's thread
      * that currently serve the client request.
      */
      public ServerTaskThread
        getTaskThread ( String signature){
              synchronized(taskHashtable){
                 return (ServerTaskThread)
                    taskHashtable.get (signature);
              }
          }

    }
```

Listing 2

```
    /**
     * Communicator is an implementation of client-server
     * communication between a test loader
     * and the main JDTS application. The client uses
     * this class in order to make the requests
     * to the server. When the request is ready to be sent,
     * the communicator attaches the client ID
     * to the request and than send it to a Sender class
     * that communicates with the server using HTTP.
     */

    public class Communicator{


    /** Communicates with the main JDTS application test
    provider. Retrieves current executing test.
     * @return Test object
     */
    public Test getCurrentTest() {
      return getTest("/getCurrentTest/");
    }

    /** Communicates with the main JDTS application test
    provider. Retrieves next test to execute.
     * @return Test object
     */
    public Test getNextTest() {
```

```
    return getTest("/getNextTest/");
}

/** Get the test based on the command specified
 * @return Test object
 */
protected Test getTest(String command) {
  try {
    /**
    * client sends get next test command to the server,
    * through the client class,
    * using the getSignature() function, that retrieves the
    * ID from the environment.
    */
   byte[] bytes = client.getNextTest(command +
     getSignature());
     .
     .
     .
     .
     .
}

/** Reports results of the test execution
 * @param Test object, which should include test log
 * and execution status (test result)
 */
public void reportTestResults(Test currentTest) {
    //send test results as byte[] using the client ID.
    client.sendTestResult("/sendTestResult/" +
      getSignature(), encoder.getBytes());
}

/**
 * Retrieve the the ID from the JAD file. When calling
 * getNextApp, the server attached to the JAD file the ID
 * that comes with
 * the request. If there is no ID attached, the ID is
 * the IP address of the agent.
 * This method returns empty String in case that there is
 * not ID attached to the JAD file.
 * Any Communicator that extends this class,
 * probably overrides this function to supply
 * its own way for getting the signature from the platform
 * environment.
 **/
```

```
public String getSignature() {
MIDlet midlet = (MIDlet)agentManager.getExtender();
String signature =
    midlet.getAppProperty("Bundle-Signature");

if (signature == null){
    return "";
        }
        else{
            return signature;
        }
}

/** Request to perform a communication action in addition
 * to the actions explicitly defined in the interface.
 * @param actionName is the name of the action to be
 * performed. The implementation filters actions according
 * to action name.
 * @param actionObjects is the array of objects this
 * action might use. The exact type of this objects needs
 * to be known
 * to both class who calls the method and to the
 * Communicator implementation
 */

public Object performAction(String actionName,]
  Object[] actionObjects) {

Object[] obj = new Object[1];

if (actionName.equals("ShowTestDescription")){
      return showTestDescription((String)
         actionObjects[0]);
}

else if (actionName.equals
  ("ShowTestDescriptionString")){
      obj[0] = actionObjects[0];
      return showTestDescriptionString
```

```
(obj, (String)actionObjects[1]);
  }

  else if  (actionName.equals
    ("ShowCombineTestDescription")){
        obj[0] = actionObjects[1];
        return ShowCombineTestDescription
          ((String)actionObjects[0], obj);
  }

  else if (actionName.equals
    ("RecordedShowCombineTestDescription")){
        obj[0] = actionObjects[1];
        return RecordedShowCombineTestDescription
          ((String)actionObjects[0], obj);
  }

  else
      return null; //unknown action
}


/**
 * This set of functions create the desired requests
 * to the be sent to the JDTS server.
 * The client ID is attached to each request.
 */
private String ShowCombineTestDescription
  (String descFileAndButtons, Object[] outputString) {

System.out.println
  ("Communicator.ShowCombineTestDescription:" +
    descFileAndButtons);

  byte[] bytes = client.sendCommand
  ("/ShowCombineTestDescription/" + descFileAndButtons +
  "/" +  getSignature(), outputString);

  if (bytes != null){
    String testResult = new String(bytes);
    return testResult;
  }
  else
    return null;
}

private String RecordedShowCombineTestDescription
  (String descFileAndButtons, Object[] outputString) {
```

```
System.out.println
  ("Communicator.RecordedShowCombineTestDescription:" +
  descFileAndButtons);

byte[] bytes = client.sendCommand
  ("/RecordedShowCombineTestDescription/" +
  descFileAndButtons +  "/" +  getSignature(),
  outputString);

if (bytes != null){
    String testResult = new String(bytes);
    return testResult;
  }
  else
    return null;
}


/** Implementation of show test description request
*/
private String showTestDescription(String description) {

System.out.println
  ("Communicator.showTestDescription:" + description);

byte[] bytes = client.sendCommand("/showTestDescription/
   + description +  "/" +  getSignature(), null);

if (bytes != null){
    String testResult = new String(bytes);
    return testResult;
}
else
    return null;
}


private String showTestDescriptionString
  (Object[] testInfo, String buttonsArray) {
byte[] bytes = client.sendCommand
  ("/showTestDescriptionString/" + buttonsArray +  "/" +
  getSignature(), testInfo);

if (bytes != null){
    String testResult = new String(bytes);
    return testResult;
  }
  else
    return null;
```

```
}

}
```

Listing 3

```
/**
* Implements the creation of the JAD file.
*/
public class HttpServer extends BaseHttpServer {

    /**
     * BaseHttpServer extender applicable for MIDP project
     */

    //JAM TAGS (default values set)
    private String profile = "MicroEdition-Profile";
    private String configuration = "MicroEdition-
Configuration";
    private String appURL = "MIDlet-Jar-URL";
    private String appSize = "MIDlet-Jar-Size";
    private String appVersion = "MIDlet-Version";
    private String appName = "MIDlet-Name";
    private String appVendor = "MIDlet-Vendor";
    private String midlet = "MIDlet-1";
    private String useOnce = "Use-Once";
    private String bundleSignature = "Bundle-Signature";
    private String permission = "MIDlet-Permissions";


    public HttpServer() {
    super();
    ALTERNATIVE_NEXT_APP = "getNextApp.jad";
    loadTagNames();
     }



    /** Runtime generation of JAM file)
     */
    public synchronized ByteArrayOutputStream
generateJAM(String host,
    int port,
    String next_app,
    long length,
```

```
        String mainClass,
        String jarLocation,
        String protocolPermissions,
        String agentMonitorID,
        String signature,
        ServerTaskThread task) throws
          UnsupportedEncodingException {

      ByteArrayOutputStream bos = new ByteArrayOutputStream();
      PrintWriter pw = new PrintWriter(new
    OutputStreamWriter(bos, "ISO8859_1"));

      File jar_path = new File(jarLocation, next_app);
      long jarsize = jar_path.length();

      pw.println(appURL + ": http://"
           + host
           + ":" + port + "/" + next_app);
      pw.println(appSize + ": " + jarsize);
      pw.println(configuration + ": CLDC-1.0");
      pw.println(profile + ": MIDP-1.0");
      pw.println(appName + ": TestSuite");  // needed by JAM
      pw.println(appVersion + ": 1.0");
      pw.println(appVendor + ": Sun Microsystems, Inc.");
      pw.println(midlet + ": TestSuite,, " + mainClass);
      pw.println(useOnce + ": yes");
      pw.println(permission + ": " + protocolPermissions);
      //the unique ID is sent to the client.
      pw.println(bundleSignature + ": " + agentMonitorID);

      //add meta-info: jad properties specified for the test
      try {
          Hashtable metaInfo =
    getTestProvider().getCurrentAppInfo(signature);
          String[] jadprops = (String[])
    metaInfo.get("JadParameters");
          if (jadprops != null) {
        //write these props to the jad
        for (int i = 0; i < jadprops.length; i++)
            pw.println(jadprops[i]);
          }
      } catch (Exception e) {
          e.printStackTrace();
          System.out.println("Failed to add jad parameter.
    [IGNORED]");
      }

      pw.close();
```

```
  task.setResponseProperty("Content-Type",
"text/vnd.sun.j2me.app-descriptor");
  task.setResponseProperty("Content-Length", "" +
bos.size());

  return bos;
    }

}
```

Listing 4

```
/**
 * Agent launcher extender compatibile with MIDlet.
 * MIDPRunner is an agent launcher for the MID
 * Profile platform.
 * When the AMS completes the download
 * of the application (JAR file),
 * it addresses the JAD file to get the class name
 * (the MIDlet name) to be executed.
 */

public class MIDPRunner extends MIDlet implements Extender
{

     private CLDCRunner runner = null;
     private String bundleID = null;

     public void startApp() {
    /**
    * this is the way get the client ID from the enviroment
    * (from the JAD file)
    */
        bundleID = getAppProperty("BundleID");
    .
    .
    .
    }

    public void pauseApp(){}
    public void destroyApp(boolean unconditional) {}

 /**
  * Returns MIDPRunner extender object as a MIDlet.
  * In order to get environment settings from other class,
  * you need to get access to this MIDlet.
  */
    public Object getRunnerExtender() {
```

```
        return (MIDlet) this;
    }

    /** Provides MIDP specific way to terminate agent
     */
    public void terminateAgent() {
        notifyDestroyed();
    }
}
```

Listing 5

```
/**
* This class is the implementation of the
* thread 44 (Fig. 3). It is used by the server
* to carry out the client request tasks.
* According to the client request, this thread
* communicates with the corresponding
* JDTS component and supplies the client request.
* Once the thread is started, the main thread of the
* server returns to listen to new requests from other
* clients, and the thread handles communication with
* the current client, until the current request has
* been fulfilled.
*/

public class ServerTaskThread extends Thread {
    /**
    * Default get next app.
    * "Thread" is a standard Java class.
    */

    protected final String NEXT_APP = "getNextApp";
    /** Alternative NEXT_APP STRING. Can be adapted per
     * platform
     */
    protected String ALTERNATIVE_NEXT_APP = "getNextApp";
    protected final String NEXT_TEST = "getNextTest";
    protected String NEXT_CMD = "/getNextCommand";
    protected String CURRENT_TEST = "getCurrentTest";


  /**
  * Sets some variable for later execution.
  */
    public ServerTaskThread(String ipAddress, Socket conn,
      BaseHttpServer httpServer)  {
        this.ipAddress = ipAddress;
```

```
        jarSourceDir = httpServer.getJarSourceDir();
        mainClass = httpServer.getMainClass();
    }

    /**
    * This function sets the initial variable, and prepares
    * it to serve the client request.
    * The thread 44 parses the client request, retrieves the
    * client ID, and checks at the main server
    * if it is to be serve a client-specific request.
    */
      public void processRequest(){
          String line, method, url, version;
          StringTokenizer cutter;
          try{
              in = new
                DataInputStream(conn.getInputStream());
              raw = conn.getOutputStream();
              out = new PrintWriter(new
                OutputStreamWriter(raw, "ISO8859_1"));

              if (!httpServer.getBatchMode()){
                  NotifierDialog.disposeNotification();
              }

              line = in.readLine();

              if (line == null) {
                  verboseln
                     ("Connection failure. Retrying.");
                  return;
              }

              verboseln("got new request: " + line);
        if (line.indexOf("getNextApp") != -1){
          getNextAppReq = true;
        }
        else {
          getNextAppReq = false;
        }

              cutter = new StringTokenizer(line);
              method = cutter.nextToken();
              url = cutter.nextToken();
              version = cutter.nextToken();

              readHeaders(in);

              if (method != null && url != null && version
```

```
                != null) {
                  try {
                      try {
                          request = new URL(url);
                      } catch (MalformedURLException e) {
                          request = new URL("http", host,
                              port, url);
                      }
              //set the client signature.
                      signature = setSignature(ipAddress,
                          request.toString());

                      if (method.equals(GET)) {
                          taskName = HANDLE_GET;

                          httpServer.putTaskThread
                              (signature, this);

                      } else if (method.equals(POST)) {
                          taskName = HANDLE_POST;

                          httpServer.putTaskThread(signature,
                          this);
                      } else {
                          verboseln
                              ("unknown request method: " +
                               method);
                          sendDiagnostics
                              (HTTP_BAD_METHOD, out);
                      }
                  } catch (MalformedURLException mue) {
                      System.out.println
                          ("Http server error: " + mue);
                      sendDiagnostics
                          (HTTP_BAD_REQUEST, out);
                  }
              }
            }catch (IOException ioe){
                ioe.printStackTrace();
            }
        }

      /**
       * The main server class, after assuramce
       * that all the parameters for the request
       * were set, calls this function
       * to start processing the request.
       */
         public void run(){
```

```
            try{
                if (taskName.equals(HANDLE_GET)){
                    handleGet(request, out, raw, in);
                }
                else if (taskName.equals(HANDLE_POST)){
                    handlePost(request, in, out, raw);
                }
                else{
                    System.out.println("Http server error");
                    sendDiagnostics(HTTP_BAD_REQUEST, out);
                }
            }catch (Exception e){
                e.printStackTrace();
            }
            cleanUp();
        }


        /**
       * This function processing HTTP GET requests.
       * The request is retrieved from the HTTP request line.
       * Then, according to the type of command
       * the coresponding functionality is being used.
       */
        protected void handleGet(URL request, PrintWriter out,
          OutputStream raw, DataInputStream in) throws
           IOException, Exception {

            /*
             *there is no need to retrieve content from the
             * client, because this method handles
             * GET requests.
             */
            String ref = (request.getRef() == null) ? ""
             : "#" + request.getRef();
            String path = request.getFile() + ref;

            if (path.startsWith(testRoot + NEXT_APP)
               || path.startsWith(testRoot +
              ALTERNATIVE_NEXT_APP)) {

                verboseln(path);

                String next_app = null;
                try {
           //request for next bundle.
                    next_app =
                     testProvider.getNextApp(signature);
```

```
                }
                catch (RuntimeException e) {
                    e.printStackTrace();
                    System.out.println
("Unable to provide next bundle possibly due to a problem
on the Client side");
                    System.out.println
("Client notified to stop the execution");
                    sendDiagnostics(HTTP_NOT_FOUND, out);
//notification to Client impl that no more bundles
available.
                    return;
                }

                if (next_app == null) {
                    sendDiagnostics(HTTP_NOT_FOUND, out);
//notification to Client impl that no more bundles
//available.
                    return;
                }

                if (next_app.length() == 0) {
                    //notification to Client impl that bundles
                     // currently unavailable.
                    //Client should wait and retry again
                    sendDiagnostics(HTTP_UNAVAILABLE, out);
                    return;
                }

                //check tags
                httpServer.checkTags();

                verboseln("next_app: " + next_app);
                File jar_path = new File(jarSourceDir,
                 next_app);
                long length = jar_path.length();

                String agentMonitorID =
                 getAgentMonitorID(request.toString());

        //call for creating JAD file, and downloaded it
                ByteArrayOutputStream bos =
                 httpServer.generateJAM(host,
                    port, next_app, length,
                    mainClass,jarSourceDir,
                    protocolPermissions, agentMonitorID,
                    signature, this);

                    sendDiagnostics(HTTP_OK, out);
```

```
                raw.write(bos.toByteArray(), 0,
                   bos.size());
                raw.flush();

        }
    // cmd "get next test"
        else if (path.startsWith(testRoot + NEXT_TEST)) {
            verboseln(NEXT_TEST);
      //get the next test from the bundle that belong to
      //this client.
            byte[] args =
               testProvider.getNextTest(signature);

           //save next_test as a currentTest for
           // subsequent calls
           // currentTests.put(signature,
           if (args == null){
                httpServer.removeCurrentTest(signature);
           }
           else{
                httpServer.setCurrentTest(signature,
                   args);
            }

            setResponseProperty("Content-Length",
            "" + (args == null ? 0 : args.length));
            sendDiagnostics(HTTP_OK, out);
            if (args != null) {
                raw.write(args);
                raw.flush();
            }

            // cmd "get current test"
        } else if (path.startsWith(testRoot +
           CURRENT_TEST)) {
              verboseln(CURRENT_TEST);
             //get the current test from the bundle that
             //belongs to this client.
            byte[] currentTest =
               httpServer.getCurrentTest(signature);

            if (currentTest != null) {
                setResponseProperty("Content-Length",
                "" + (currentTest == null ? 0 :
                currentTest.length));
                sendDiagnostics(HTTP_OK, out);

                if (currentTest != null) {
                    raw.write(currentTest);
```

```
                    raw.flush();
                }
            }
            else {
                throw new
                IllegalStateException
("getCurrentTest called prior to the getNextTest");
            }
        }
        // assume cmd is a request for download of
        //the JAR file to the client.
        else {

            File file = new File(jarSourceDir, path);
            verboseln("file download: " + file);

            if (file.isFile() && file.canRead()) {
                try {
                    FileInputStream fis = new
                       FileInputStream(file);
                    DataInputStream fin = new
                       DataInputStream(fis);
                    byte[] buffer = new
                       byte[(int)file.length()];
                    fin.readFully(buffer);
                    fin.close();
                    setResponseProperty
                       ("Content-Type",
                       "application/java-archive");
                    setResponseProperty("Content-Length",
                       "" + file.length());
                    sendDiagnostics(HTTP_OK, out);
                    raw.write(buffer);
                    raw.flush();
                    return;
                } catch (IOException ioe) {
                    System.out.println("error: " +
                       ioe.getMessage());
                    sendDiagnostics(HTTP_SERVER_ERROR,
                       out);
                }
            }
            sendDiagnostics(HTTP_NOT_FOUND, out);
        }
    }

    /**
    * This function processes HTTP POST requests.
```

```
     * The request is retrieved from the HTTP request line.
     * Then, according to the type of command
     * the corresponding functionality is used.
     */
5    protected void handlePost(URL request, DataInputStream in,
     PrintWriter out, OutputStream raw) throws IOException {

          String ref = (request.getRef() == null) ? "" : "#"
            + request.getRef();
10        String path = request.getFile() + ref;

          if (processCommand(request,postData,out,raw,path))
          {
              verboseln(path + " processed");
15        }
          else
              sendDiagnostics(HTTP_NOT_FOUND, out);
          }

20

        /**
         * These are common actions for POST http methods
         **/
        private boolean processCommand(URL request,byte[]
25       buf,PrintWriter out,OutputStream raw,String path)
          throws IOException {

            // cmd "send test results"
            if (request.getFile().startsWith(testRoot +
30            "sendTestResult")) {
                verboseln("sendTestResult");

                //String signature = getSignature (path);
                testProvider.sendTestResult(buf, signature);
35              sendDiagnostics(HTTP_OK, out);
                return true;
            }
            //handle storeEventSequence cmd.
            else if (getUIService("storeEventSequence",
40            path, testRoot, out, raw, buf)){
              verboseln(path + " processed");
              return true;
            }

45           //handle showTestDescriptionString cmd.
             //For TCK tests.
             else if (getUIService
               ("showTestDescriptionString", path, testRoot,
                out, raw, buf)){
```

```
                verboseln(path + " processed");
                return true;
            }

        //handle ShowCombineTestDescription cmd.
        // For J2SE tests.
        else if (getUIService("ShowCombineTestDescription",
          path, testRoot, out, raw, buf)){
            verboseln(path + " processed");
            return true;
        }

        //handle compareJ2SEOutputString cmd.
        //For J2SE tests.
        else if (getUIService("compareJ2SEOutputString",
         path, testRoot, out, raw, buf)){
            verboseln(path + " processed");
            return true;
        }
        else
            return false;
    }


/*
* this method is called when the former task wasn't
* ended normally, like VM_EXIT. When the next
* "getNextApp.jad" request
* arrives, JDTS checks that the last task had
* finished normally, and if doesn't then it calls
* stopTask(), to clear the viewer in case of an UI test,
* and calls clean up to close all the streams.
*/
    public void stopTask(){
        try{
            if (automationManager.viewer != null) {
                automationManager.viewer.stop();
                automationManager.viewer = null;
            }
            cleanUp();
        }catch (Exception e){
            e.printStackTrace();
        }
    }

/**
 * This function set to the client its unique ID. In case
 * that the IP is used is a local address (127.0.0.1),
 * JDTS assigns an ID to the client, Otherwise the IP
```

```
     * address IS be used as an ID, to which it is
     * concatenated with another string, which may be an ID
     * attached to the first client request by some outside
     * party.
     */
        public String setSignature(String ipAddress,
          String path){
    //happens only on the first time that a MIDP client
    //connects to JDTS using the host as a "localhost"
    //string.
            if (ipAddress.equals("127.0.0.1") &&
             getNextAppReq == true){
                httpServer.setLocalhostFlag();
                String ID =
                   String.valueOf(httpServer.getNextID());
                   verboseln("ID = " + ID);
                   return ID;
            }
    //last request was made using the localhost address and
    //the localHost flag was set,
    //That means that the requests from the client
    //use a signature that was given to the JDTS server
    //and there is no significance to the IP address.
            if (httpServer.getLocalhostFlag()){
                return getAgentMonitorID(path);
            }

            //common case.
            String agentMonitorID =  getAgentMonitorID(path);
    //get an ID that was made by some outside party.
            return ipAddress + "." + agentMonitorID;
        }

        private String getAgentMonitorID(String path){
            String lastToken = null;
            StringTokenizer st  =
               new StringTokenizer(path, "/");
            int tokensNum = st.countTokens();
            while (tokensNum > 0){
                lastToken = st.nextToken();
                tokensNum--;
            }
            try{
             //if next token is a number than the last token
             //is an ID, and an exception is not generated.
                Integer.parseInt(lastToken);
                return lastToken;
            }catch (NumberFormatException nfe){
             //default ID
```

```
            return "1";
        }
    }

/**
 * A service for UI test
 */
   private boolean getUIService(String task, String path,
     String testRoot, PrintWriter out, OutputStream raw,
     byte[] buf) throws IOException {
        try{
            synchronized(automationManager){
                verboseln("signature = " + signature);
                verboseln("ipAddress = " + ipAddress);
      //set the client ID before address the UI request.
             automationManager.setSignature(signature);
             automationManager.setIPAddress(ipAddress);

             if (task.equals("showTestDescription")){
               return automationManager.
                 showTestDescription
                 (path, testRoot, out, raw);
             }
             else if (task.equals
              ("getRecordedEventSequence")){
               returnautomationManager.
                 getRecordedEventSequence
                 (path, testRoot, out, raw);
               }
               else if (task.equals
                 ("storeEventSequence")){
                 return
                 automationManager.storeEventSequence
                      (path, testRoot, out, raw, buf);
               }
               else if (task.equals
                 ("showTestDescriptionString")){
                 return automationManager.
                 showTestDescriptionString
                 (path, testRoot, out, raw, buf);
               }
               else if (task.equals
                 ("ShowCombineTestDescription")){
                      return automationManager.
                       ShowCombineTestDescription
                      (path, testRoot, out, raw, buf);
               }
               else if (task.equals
                 ("compareJ2SEOutputString")){
```

```
                    return automationManager.
                        compareJ2SEOutputString(path,
                        testRoot, out, raw, buf);
                }
                else
                    return false;
            }
        }catch (Exception e){
            e.printStackTrace();
            return false;
        }
    }


    private void cleanUp(){
        try{
            /*
             *In case that the cleanUp call made from a
             *situation of TIMEOUT, meaning that the client
             *has died, there is no need
             *for the following code
             */
            if (status == 2){  //Client is ALIVE
             /*
              * Read and block until the end of the input
              * stream, so we know that
              * the client application got the data, not
              * just the low level TCP.
              */
                try {
                    for (; ; ) {
                        if (in.read() == -1) {
                            break;
                        }
                    }
                } catch (IOException se) {
                    // do nothing
                }
            }
            in.close();
            out.close();
            raw.close();
            conn.close();
        }catch(IOException ioe){
            ioe.printStackTrace();
        }
    //Remove the thread 44 from the active threads list.
        httpServer.cleanUp(signature);
```

```
    }
}
```

Listing 6

```
/**
 * Extender is a class that enables an agent to be linked
 * with platforms that require their main application
 * class to extend their system class,
 * such as Applet or MIDlet.
 * Extender enables agent to delegate platform specific
 * commands to it.
 *
 * @author Victor Rosenman
 * (SUN ISRAEL DEVELOPMENT CENTER)
 *
 */

public interface Extender {
/** Enables extender to use platform specific way
 * of application termination
 */
public void terminateAgent();

/** Retrieves a reference to platform class, which
 * the main application class extends.
 * The agent can provide access to the test program to
 * the main application class in the context
 * in which it is running.
 * @return an object, which can be casted to the system
 * class, which extender class extends
 */

public Object getRunnerExtender();

}
```

Listing 7

```
*/
if (path.startsWith(testRoot + NEXT_APP)
 || path.startsWith(testRoot + ALTERNATIVE_NEXT_APP)) {
  verboseln("path = " +  path);

  String next_app = null;
```

```
    try {
      next_app = testProvider.getNextApp(signature);
}
.
.
.
```

Listing 8

```
/* Provides the next bundle associated with the
 * current ID. If getNextApp is called
 * twice consecutively with the same ID, an error
 * is registered and ignored 3 times. On the
 * fourth call an exception is thrown. We assume
 * something went wrong on the client side.
 */

public synchronized String getNextApp(String ID) {

BundleDescriptor descriptor = etBundleDescriptor(ID);
// first check whether there are tests left in this app
if ((descriptor != null)
    && (descriptor.executingBundle != null)
    && (!descriptor.executingBundle.isEmpty())) {

// previous test execution has not been completed
// due to failure in the client.
  if (descriptor.executingTest == null) {
    if (descriptor.atLeastOneTestRun) {

/*
 * The system had to restart, so getNextApp
 * is called.
 * However as the previous test has normally run
 * getNextTest is called,
 * Results received Therefore we return
 * the same bundle to be used
 * during the execution. Next test will run when
 * getNextTest is called
 */
        descriptor.executingBundle.restarting();
        descriptor.nErrors = 0;
        descriptor.executingTest = null;
        descriptor.atLeastOneTestRun = false;
        return descriptor.executingBundle.getApp();
  }
  else {
//getNextApp run without invocation to getNextTest
```

```
        if (descriptor.nErrors > 1) {
        System.err.println("ERROR: "+
         (++descriptor.nErrors + 1) +
         " getNextApp calls in a row");
        descriptor.nErrors = 0;
        throw new RuntimeException(ID + ": request for
         new bundle arrived before current bundle tests
         where invoked");
        }
        else
          System.err.println("ERROR: "+
           (++descriptor.nErrors + 1) +
          " getNextApp calls in a row [ERROR GNORED]");
        }
    }
  else {
  //we need to signal VM exit, due to failure
  //in the client side.
      descriptor.executingBundle.passVMExitResult
       (descriptor.executingTest);
      descriptor.nErrors = 0;
      descriptor.executingTest = null;
    }

  descriptor.executingBundle.restarting();
  return descriptor.executingBundle.getApp();
  }

  //check intermediate condition
  if (bundles.isEmpty()) {
    // no more apps yet arrived
    // so let JAM wait
    return "";
  }

  //provide new bundle from the stack that holds all
  //bundles that are non-associated
  TestBundle next = (TestBundle) bundles.elementAt(0);

  if (next != null) {
  // we have more bundle to execute!
  // Create bundle descriptor object and
  // associate it with the caller ID

    descriptor = new BundleDescriptor((TestBundle)
     bundles.remove(0));
    associateBundle(descriptor, ID);

  //signal execution starting
```

```
      descriptor.executingBundle.starting();

    //provides app name of this bundle
      return descriptor.executingBundle.getApp();
  } else {
    //end of execution process reached
    //null returned - to signal the end state
       notifyListeners();
       return null;
    }
  }
```